Chapter Four: Contents

(Route Planner – 11 June 2001 – LA-UR-00-1767 – TRANSIMS 2.0)

Chapter Four: Figures

Chapter Four: Tables

Chapter Four—Route Planner

1. INTRODUCTION

1.1 Overview

As its name implies, the Route Planner module generates routes for travelers. Each traveler, including transit drivers, itinerant travelers, and truck drivers, receives an individual travel plan. Once the plans are generated for all travelers, they are simultaneously executed in the Traffic Microsimulator.

Constraining the routes between different locations are

1) the transportation network, which represents the metropolitan region being studied, and

2) the preferences of individual travelers.

Information about each traveler's activities (contained in Chapter Three: (*Activity Generator*)) is used to create trip requests. A trip request consists of three parts:

- the origin and destination of the trip,
- the ranges for the preferred starting and ending time and duration, and
- the travel mode choice.

Given in the form of a string of characters, the travel mode choice defines the allowed modes of travel and their order. The Mode Preference File (configuration file key: `MODE_MAP_FILE`) defines the mapping between mode choice numbers (as used in the activity file) and mode choice strings (as used by the Route Planner). Additional information about the vehicles that a particular traveler may use is contained in the Vehicle File (configuration file key: `VEHICLE_FILE`). Fig. 1 shows the data flow the Route Planner uses to generate plans for individual travelers.

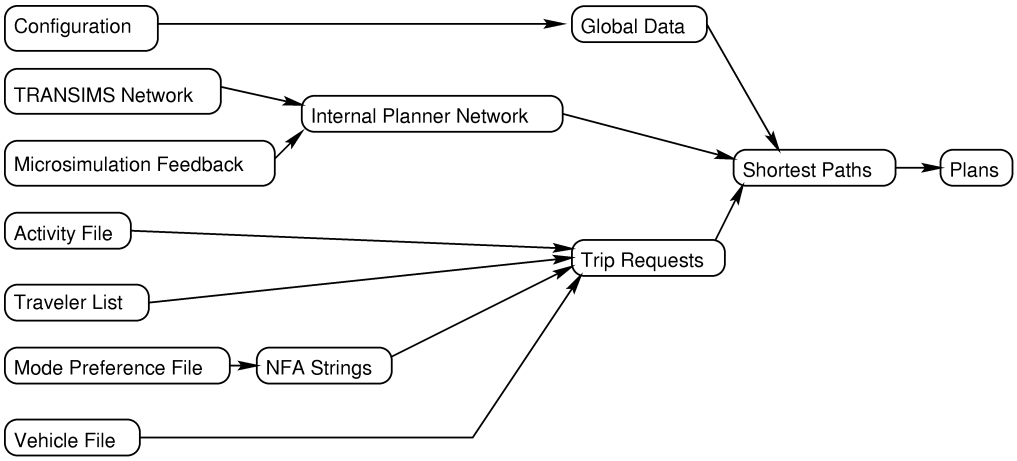

Fig. 1. Data flow diagram that shows how the TRANSIMS Route Planner generates travel plans for travelers.

1.2 TRANSIMS Network

The TRANSIMS Network provides information about the streets, intersections, signals, parking lots, activity locations, and transit stops in a road transportation network. This information is used to construct the Route Planner Internal Network. The internal network is time dependent—that is, travel on a link may incur different delays at different times of the day. The information about delays on links is derived from the Traffic Microsimulator output and provided in the Feedback File (configuration file key: `ROUTER_LINK_DELAY_FILE`), which specifies the mean delays on each link over 15-minute intervals. If the delay for a particular interval is not given, the free speed delay is used. More information about link delays may be found in Section 3.7.1.

The Route Planner's core is Dijkstra's shortest-path-finding algorithm, with extensions for time-dependent delays and paths constrained by travel mode. The internal network and trip requests are given to the path-finding algorithm, which creates routes and outputs them in the form of plans.

1.2.1 Route Planner Major Input/Output

Fig. 2 shows the Route Planner's major inputs and outputs. The major inputs to the Route Planner are

- transit routes and schedules,
- the activities list,
- the TRANSIMS multimodal network,
- the vehicle file, and
- link travel times.

The major outputs of the Route Planner are

- the plan list, and
- the anomalous activity list.

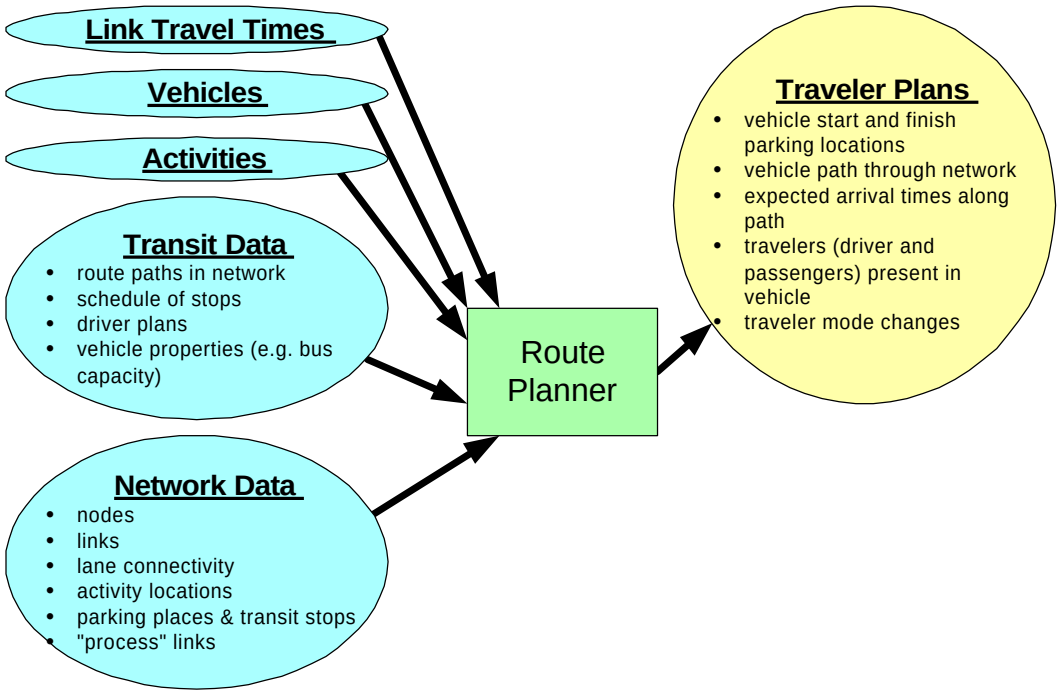

Fig. 2. The major input to the Route Planner includes the following data: (1) TRANSIMS Network, (2) activities, (3) transit, and (4) vehicle information from the synthetic population data.

1.2.2 Parallelization

To increase execution speed, the Route Planner may be parallelized (run on several processes of the same machine) and distributed (run on several machines). These techniques may be combined, allowing the Route Planner to take full advantage of a cluster of multiprocessor machines.

Threads enable the parallel execution of several copies of the path-finding algorithm on a shared memory machine. Each planning thread uses the same copy of the network to create plans and trip requests for different households. The plans created by the different threads are written to the plan file (configuration key `PLAN_FILE`). The number of threads that is used is controlled by the configuration key `ROUTER_NUMBER_THREADS`. If this key has a value of 0, threading is disabled completely. A positive value indicates the number of route planning threads to use. In addition, one thread responsible for reading households from the activity file and one thread responsible for writing plans to the plan file are used. Therefore, if threading is enabled, there are a minimum of three threads

(one input thread, one output thread, and one planning thread). In general, the number of planning threads should be equal to the number of available processors. This will effectively overlap computation with I/O.

Activities are assigned to threads using a round-robin approach; so, for the same activity list, each thread is always given the same households to plan. This is important for repeatability, so that the same random numbers are used in different runs of the Route Planner.

When running on several computers, several instances of the Route Planner may run concurrently. In this case, the Route Planner is started on each machine with the command:

```
Router <configuration file> <rank>
```

where `rank` is an integer starting at 0, identifying the processor on which this copy of the Route Planner is executing.

The household file, completed household file, plan file, and anomalous activity file are unique for each process and are formed by appending *.txx* to the appropriate filename, where *xx* is the rank expressed as a two-digit base 26 number (i.e., the sequence is AA, AB, …, AZ, BA, BB, …). Households are assigned to processes by creating the appropriate household file. The utility *MakeHouseholds* can be used to create appropriate household files. See Section 4 for more information.

If the configuration file key `ROUTER_COMPLETED_HOUSEHOLD_FILE` is set, household IDs will be written to this file as they are completed. This allows restart capability, as any households whose IDs are in this file need not be replanned. Because each process has its own completed household file, individual processes may be restarted independently. Care must be taken to avoid writing over the partial plan and problem files.

2. ROUTE PLANNER DESCRIPTION

2.1 Overview

The Route Planner computes the "shortest" path, subject to mode constraints, for each traveler in the system. Each link within the transportation network has a cost associated with it. Accordingly, the shortest path can be interpreted as least cost, for some generalized meaning of cost. Constraints are provided by criteria such as mode preferences for different legs of the trip.

Costs for a link can be computed simply with input, such as an estimated time delay.

There are also more sophisticated ways to calculate costs. For example, they can be calculated based on several variables, including time delays and the actual monetary costs of a link. More abstract variables can be used, such as a penalty for traveling through construction areas, and traveler demographics, such as household income level.

2.2 Distinguishing Features

The Route Planner has three distinguishing features.

2.2.1 Individual Plans

Plans are computed for each individual traveler in the population, based on that individual's activity demands and preferences. Such computations enable each traveler to have an individualized view of the transportation system. Accordingly, costs associated with links in the network are computed separately for each traveler.

2.2.2 Per Link Time-Dependant Delay

Link costs are computed in a time-dependent manner that can account for time delays resulting from actual travel conditions, such as peak-hour congestion. These delays are fed back from the Microsimulation into the Route Planner, enabling routes to be changed for individual travelers.

2.2.3 Travel Mode Constraints

The Route Planner abides by any mode preferences contained in the activity files. Thus, if the activity files specify that a traveler will walk, then take a car, and then walk again between two desired activities, the Route Planner will produce a plan (if feasible) that ensures these modes are used in this sequence.

2.3 Terminology

2.3.1 Traveler Plan

A *traveler plan* consists of a set of trips that carries the traveler through his or her desired activities. A *trip* consists of a set of contiguous legs. Activities of a given duration at a specific location may be separate trips. A *leg* consists of contiguous nodes and links that are traversed with a single travel mode. For example, a trip may consist of three legs:

- walking,
- transit, and
- walking.

A traveler plan could consist of:

- a home activity,
- a trip from home to work
- a work activity,
- a trip from work to shopping,
- a shopping activity,
- a trip from shopping to home, and
- a home activity.

2.3.2 Transit Vehicle

From the point of view of the Route Planner, a *transit vehicle* is considered to be any vehicle that makes scheduled stops along a predetermined route. Buses, trains, and streetcars are all considered transit vehicles, whereas a taxi would not be considered a transit vehicle.

2.3.3 Trip Request

A request for travel to be planned by the Route Planner, a *trip request* consists of a starting location, a destination location, a start time, end time, duration constraints, and a mode string.

2.3.4 Mode String

A mode string contains a list of travel modes that must be used in the order given along the path from source to destination.

2.4 Travel Modes

There are twelve individual modes available within TRANSIMS. The modes and their corresponding mode letter are show in Table 1. Bike mode is routed at a faster speed on the walk network. Transit mode allows travel on any type of mass transit system (bus, rail, streetcar, or trolley) and allows walking in between transit routes. This allows transfers between different types of transit that may not use the same transit stop.

Magic mode is an unrouted mode. For magic moves, a walk plan is generated whose start and stop times are taken from the times given in the activities. Its intended use is to enable the use of travel modes that are not supported by the Route Planner and/or the Traffic Microsimulator, such as school busses. The mode string wcwxw, where x is one of b, l, g, p, y, s, or t, is used for park-and-ride. Park-and-ride is not currently supported.

Any mode string consisting of the mode letters in Table 1 can be planned. Some are meaningless because they will never produce paths (e.g., cb—because a traveler must walk from a parking location to a bus stop).

Table 1. Currently recognized travel mode letters.

Mode	Mode Letter
Walk	w
Bike	i
Car	c
Bus	b
Light Rail	l
Regional Rail	g
Rapid Rail	p
Trolley	y
Street Car	s
Transit	t
Magic Move – School Bus	K
Magic Move -- Other	k

The mapping between the mode strings used by the Route Planner and the mode numbers used in the activity file is given by the mode map file (configuration file key `MODE_MAP_FILE`). Each line in this file contains a mode number and a corresponding mode string.

An example mode file containing three mappings is shown below.

```
1 w
2 wcw
3 wtw
```

2.5 Trip Requests

2.5.1 Generating Trip Requests from Activities

The activity, vehicle, and mode files are used to generate trip requests, which are then planned. In the activity file (configuration file key: `ACTIVITY_FILE`), travelers' mode preferences are given by integers. Their meaning is defined by the mode file (configuration file key: `MODE_MAP_FILE`), which gives the correspondence between these integers and mode strings used by the Route Planner.

The Route Planner uses the household file (configuration file key: `ROUTER_HOUSEHOLD_FILE`) to determine the travelers for which plans should be generated. If this value defines a file that can be opened, then only the travelers belonging to households whose IDs are listed in this file will be planned. Otherwise, the Route Planner plans all of the travelers in the activity file. All travelers in a single household are planned together because they may share transportation or activities. The Selector/Iteration Database uses the household file as part of the feedback mechanism that enables a portion of the population to be re-planned.

Plans are generated in response to trip requests for a traveler. Trip requests come from the activity file. For every traveler, each pair of consecutive activities at different locations generates a trip request. A trip request consists of a source activity location; a destination activity location; constraints on the start time, end time, and duration; and the travel modes that are allowed. A trip request is satisfied by a plan, in the form of a trip made up of unimodal legs. Travel plans are separated by activity plans.

The activities of each traveler are split into legs that define either activities (activity legs), or travel (transportation legs). Activity legs begin and end at the same activity location. Transportation legs begin and end at different activity locations. The activity legs are not planned, and are written into the plan file using the times from the activity file. Travel plans are created for the transportation legs. If a transportation leg is multimodal, it is further split up into unimodal sections, which are planned as separate legs of a trip.

If a planned trip uses a car, the vehicle file (1) is examined to find the location of the car, and (2) the trip is split. The first mode string ends with the last symbol before `c`, and the destination of the first part of the trip is the parking location where the car is located.

The second part of the trip starts there (with mode `c`) and ends at the original trip's destination. The two parts are planned separately then written out consecutively in the plan file.

2.5.2 Time Priority

Each activity has a time priority field that describes which of start time, end time, and duration is important for that activity. The Route Planner uses this information to fit transportation legs in between activity legs. Table 2 describes the various values of the time priority field.

Table 2. Description of time priorities.

Time Priority	Important Time Start	Stop	Duration
0			
1	X		
2		X	
3	X	X	
4			X
5	X		X
6		X	X
7	X	X	X

The following describes how the Route Planner uses the time priority field to determine the start time, stop time, and duration of activity legs.

The start time of an activity is mainly determined by the end time of the preceding transportation leg (PTL). If there is no PTL (because this is the first activity for the traveler) or the PTL ends prior to the lower bound of the start time specified for this activity, the start time is taken from the distribution given in the activity file. If the activity time priority doesn't specify start time (priorities 0,2,4,6), the start time of the activity is the maximum of the end time of the PTL and the lower bound of the start time of this activity.

If the activity time priority does not include start time (priorities 1,3,5,7) and the PTL end time is prior to the lower bound of the activity start time, then pick a start time from the distribution. If the PTL end time is greater than the activity start time upper bound, then the PTL start time is decreased, if possible, so that the PTL end time is equal to the activity start time upper bound. This is only done if the constraints on the previous activity are not violated. Otherwise, the start time is the arrival time of the PTL.

Next, the duration and stop time of the activity must be determined. Of these two, if only duration is specified by the time priority (priorities 4,5), a duration is picked from the distribution given in the activity file. The stop time is then the start time plus duration. For all other priorities, a stop time is picked from the distribution given in the activity file. The duration is the difference between the stop time and start time. If the resulting duration is 0 or less, then the duration is changed to 1, and the stop time is changed to start time+1.

Finally, the times listed as important by the time priority are checked against the ranges specified by the activity file. An entry in the anomalous activity file is created for any time indicated by the time priority that does not fall in the proper range; however, the traveler is still planned.

2.6 Parking

Because TRANSIMS tracks the movements of each individual throughout the simulation, the Route Planner retains the location of each household's vehicles. This enables an individual from a household to drive to a parking location, walk from the parking lot to work, then return to the same parking location to retrieve the vehicle for the trip home.

Currently the Route Planner will pick a parking location adjacent to the destination activity location for the trip as the destination parking location. If there is no adjacent parking location, the Route Planner will display a warning and skip the remainder of the traveler's activities. In this case, adjacent means that there is a process link from the ending parking location to the ending activity location. This restriction will be removed in a future version of the Route Planner. If the ending activity location is adjacent to the starting parking location, then only a walk trip, from the starting activity location to the ending activity location is generated, and an entry is made in the anomalous activity file.

When an activity has a mode string of wcwtw (outbound) or wtwcw (returning), it is assumed that the car should be parked at a park and ride lot. These lots are designated as such in the TRANSIMS Network parking table (configuration file key: `NET_PARKING_TABLE`). Park and ride lots may also be used on non-park and ride trips.

2.7 Shared Rides

A shared ride is one in which a passenger travels in an automobile driven by another traveler. Currently only shared rides in which the passenger and the driver are part of the same household are supported. The driver trip request is planned as usual. Any passenger trip requests are fulfilled, after all of a household's non-passenger trips have been planned, by using information from the driver plans.

The driver and passenger trip requests are matched according to the following procedure. The trip requests for a passenger with a particular driver are listed in the order that they occur in the activity file. The driver trip requests that include the passenger are also listed in activity file order. The driver and passenger trip requests are then matched in order according to these lists. This process is repeated for every combination of driver and passenger that occurs in a household. If there are not enough driver trip requests to satisfy all of the passenger trip requests, the passenger activity is listed in the anomalous activity file with an anomaly type of *Invalid Shared Ride*. The condition where there are too many driver trip requests is not detected.

Because of interdependencies between travelers (a passenger in the morning may be a driver in the afternoon), a passenger activity may be planned before the corresponding driver activity. Room for the passenger trip is left in the plan sequence according to the desired activity times. If the driver trip is longer than expected (e.g., because of congestion), there may not be enough time between activities in the passenger plan. In this case, the activity leg following the passenger trip is shortened to accommodate the transportation leg and the activity is recorded in the anomalous activity file with the

anomaly type *Invalid Shared Ride Time*. If the passenger trip extends past the end of the upper bound of the following activity, the remaining activities for the passenger are not planned.

2.8 Anomalous Activity File

There are currently seven types of anomalous activities recognized by the Route Planner*: No Path*, *Invalid Time*, *Invalid Shared Ride*, *Invalid Shared Ride Time*, *Connectivity*, *Location*, and *Parking* (see Table 3). These data can be used by the Selector/Iteration Database module to request new activity characteristics for the traveler of the household. An error anomaly prevents the planning of the rest of the activities for a traveler, while a warning anomaly does not.

Table 3. Types of anomalies.

Anomaly Type	Number	Severity
No Path	1	Error
Invalid Time	2	Warning
Invalid Shared Ride	3	Error
Invalid Shared Ride Time	4	Subtype 1,3: Warning Subtype 2,4: Error
Connectivity	5	Warning
Location	6	Error
Parking	7	Warning

For each activity in which an anomaly is detected, a line is written to the anomaly activity file (configuration file key: `ROUTER_PROBLEM_FILE`). The first eight fields (see

Table 4) of each line are the same for each type of anomaly. These fields describe the activity for which an anomaly was detected, the trip generated for this activity, the type and subtype of anomaly detected, and the number of anomaly-specific fields remaining. If no trip was generated for this activity, then the `TripId` and `LegId` fields are set to -1.

Table 4. Anomalous activity file common fields.

Field	Description
`HouseholdId`	ID of the anomalous household.
`TravelerId`	ID of the anomalous traveler.
`ActivityId`	ID of the anomalous activity.
`TripId`	ID of the trip generated by this activity.
`LegId`	ID of the first leg generated by this activity.
`ProblemType`	Type of anomaly (See Table 3)
`Problem Subtype`	Subtype of an anomaly, type dependent.
`Number of data fields`	Number of remain fields, varies by anomaly type.

2.8.1 *No Path* Anomaly

A *No Path* anomaly takes place when a trip request cannot be satisfied because a path from the source location to the destination location which obeys the time and mode constraints could not be found. Common reasons for this anomaly include no connectivity between the source location and the destination location, and no transit vehicles running after the start time. The *No Path* anomaly includes information about the source and destination accessories, the mode, and the start time of the transportation leg. When a *No Path* anomaly is detected, no plan is generated, and the rest of the activities for this traveler are skipped. Table 5 describes the subtypes of the *No Path* anomaly.

Table **6** describes the *No Path* fields. The maximum trip length, leg length, and number of nodes searched can be set with the configuration file keys `ROUTER_MAX_TRIP_TIME`, `ROUTER_MAXIMUM_LEG_LENGTH`, and `ROUTER_MAXIMUM_NODES_EXAMINED`, respectively.

Table 5. *No Path* Subtypes

Subtype	Value	Description
`No path exists`	1	No path exists with the requested mode, at the requested time.
`Trip Length`	2	The activity starts past the end of the simulation.
`Leg Length`	3	The trip leg is too long.
`Max Nodes`	4	The maximum number of nodes has been searched.

Table 6. *No Path* fields.

Field	Description
`SourceLocation`	The source location of the anomaly trip.
`SourceType`	The source location type of the anomaly trip.
`DestinationLocation`	The destination location of the anomaly trip.
`DestinationType`	The destination location type of the anomaly trip
`Mode`	The travel mode of the anomaly trip
`StartTime`	The time the anomaly trip should start.

2.8.2 *Invalid Time* Anomaly

An *Invalid Time* anomaly occurs when the actual time used by the Route Planner does not fit within the bounds specified by the activity. The start time, end time, and duration are checked for consistency with the ranges given in the activity. A separate line in the anomalous activity file is output for each one of these times that is inconsistent. The line contains the type of the inconsistency, the lower and upper bound from the activity file, and the actual value used by the Route Planner. A plan is generated for the anomalous activity using the inconsistent times. Table 7 describes the *Invalid Time* fields.

Table 7. *Invalid Time* fields.

Field	Description
`TimeType`	The field that has the anomaly 0-Start, 1-End, 2-Duration.
`LowerBound`	The distribution lower bound.
`UpperBound`	The distribution upper bound.
`Actual`	The actual value used.

2.8.3 *Invalid Shared Ride* Anomaly

An *Invalid Shared Ride* anomaly occurs when the driver activities and passenger activities do not match up. Currently, only the condition where there are too few driver activities for the number of passenger activities is detected. When this anomaly is detected, no plan is generated for the passenger and the rest of the passenger's activities are not planned. The driver activities are planned as usual. No extra fields are output for this anomaly.

2.8.4 *Invalid Shared Ride Time* Anomaly

An *Invalid Shared Ride Time* anomaly takes place when the transportation leg for a passenger-shared ride takes longer than the time between the two surrounding activity legs. If the trip extends past the upper bound of the following activity's start time, but not past the following activity's end time, an *Invalid Shared Ride Time* entry is created in the anomalous activity file, and the rest of the passengers trip requests are planned. If the trip extends past the end time of the following activity, an *Invalid Shared Ride Time* entry is created in the anomalous activity file, and no further trips are planned for this traveler. The *Invalid Shared Ride Time* anomaly contains the arrival time of the passenger-shared ride trip, the upper bound of the start time of the following activity, and the end time of the following activity. Table 8 describes the subtypes of the *Invalid Shared Ride Time* Anomaly, while Table 9 describes the *Invalid Shared Ride Time* fields.

Table 8. *Invalid Shared Ride Time* subtypes.

Subtype	Value	Description
Driver Late	1	The driver was late, but the length of the following activity was adjusted to compensate.
Driver Very Late	2	The driver was too late to be accommodated.
Passenger Late	3	The passenger was late, but the length of the following activity was adjusted to compensate.
Passenger Very Late	4	The passenger was too late to be accommodated.

Table 9. *Invalid Shared Ride Time* fields.

Field	Description
`Arrival Time`	The arrival time of the passenger-shared ride trip.
`Start Time Bound`	The upper bound of the starting time of the activity leg following the passenger-shared ride trip.
`Stop Time`	The stop time of the activity leg following the passenger-shared ride trip.

2.8.5 *Connectivity* Anomaly

A *Connectivity* anomaly occurs when there does not exist a process link from the destination parking location to the final activity location. When this happens, a plan is still produced as this process link is not included in the output plan. Table 10 describes the Connectivity fields.

Table 10. *Connectivity* fields.

Field	Description
`Accessory Id`	ID of the destination activity location.
`Accessory Type`	Type of the destination activity location.
`Parking Id`	ID of the destination parking location.

2.8.6 *Location* Anomaly

A *Location* anomaly occurs when the source activity location or destination activity location specified in the activity file or the vehicle location specified in the vehicle file cannot be located in the TRANSIMS transportation network. Table 11 describes the *Location* fields.

Table 11. *Location* fields.

Field	Description
`Accessory Id`	ID of the accessory that cannot be found.
`Accessory Type`	Type of the accessory that cannot be found.

2.8.7 *Parking* Anomaly

A *Parking* anomaly occurs when the origin parking location and destination parking location are identical. This occurs when a drive trip is specified between two activity locations that share a parking location. A walk trip between the two activity locations is generated. Table 12 describes the *Parking* fields.

Table 12. *Parking* fields.

Field	Description
`Source Activity`	ID of the origin activity location.
`Destination Activity`	ID of the destination activity location.
`Parking Id`	ID of the common parking location.

2.9 Network Layers

The Route Planner conceptually views the network as a set of interconnected, unimodal layers (see Fig. 3). In other words, a separate layer exists for each mode letter in the mode string. At certain designated locations (which becomes nodes in the Route Planner's view of the network) in each layer, a special link, called a process link, connects one or more of the unimodal layers to another. These process links allow intermodal transitions to take place.

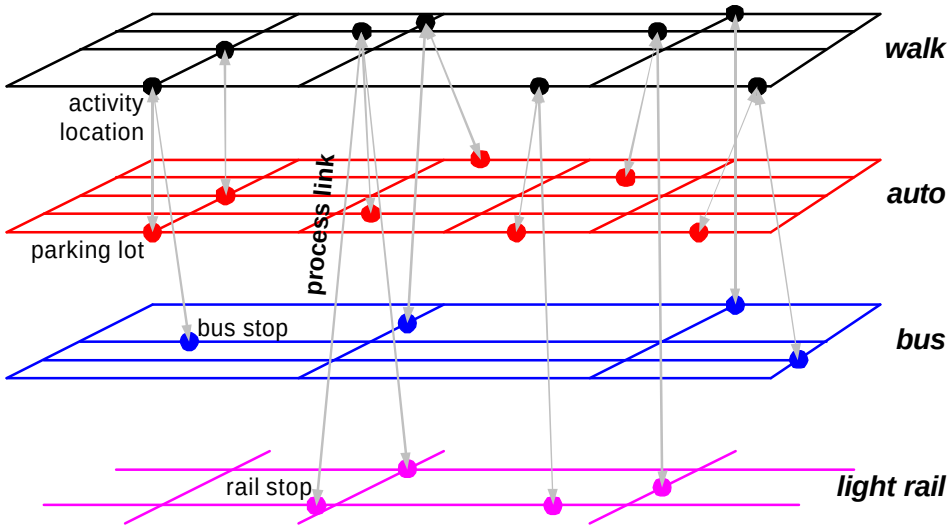

Fig. 3. A high-level depiction of the various layers used by the Route Planner. From individual traveler preferences and constraints contained in the synthetic population and activities data blocks, the Route Planner plans for trips that consist of multiple modal legs (e.g., walk-car-walk). Constructing multiple layers in which each layer can be encoded as a different unimodal network allows for the efficient calculation of trips constrained by modal sequences. Also shown are the process links connecting the unimodal networks.

The process links are considered to be part of the walking layer. The layers are constructed from the TRANSIMS Network. Delays for each link in each layer are computed by a link delay function, which is time-dependant. Link delays are further explained in Section 3.7.1.

Conceptually, layers are associated with modes of travel. In this view, there are three types of layers in the network:

1) A street layer, which consists of all links between intersections, and parking locations.

2) A walk layer, which consists of all streets that can be walked along and activity locations. However, the parking locations and transit stops that belong to the other two layers are accessible only from activity locations via process links.

3) Transit stops and links to transit layers, which can be traversed in transit (e.g., bus or light rail) modes only. There is a separate layer for each type of transit vehicle (e.g., bus and light rail), and a layer for each transit route via process links.

In Fig. 4, nodes A and B are street nodes. They correspond to original TRANSIMS Network nodes. Nodes P and Q are parking locations, whereas R and S are activity locations. Nodes C and D are transit stops. The links between layers are called process links.

Conceptually, nodes A and B appear in two different layers, even though these appearances correspond to the same TRANSIMS nodes. The reason for this is that even though we might be in the same geographic location (whether in a street or walk network), we cannot change from the street to the walk network without visiting an activity location and using a process link.

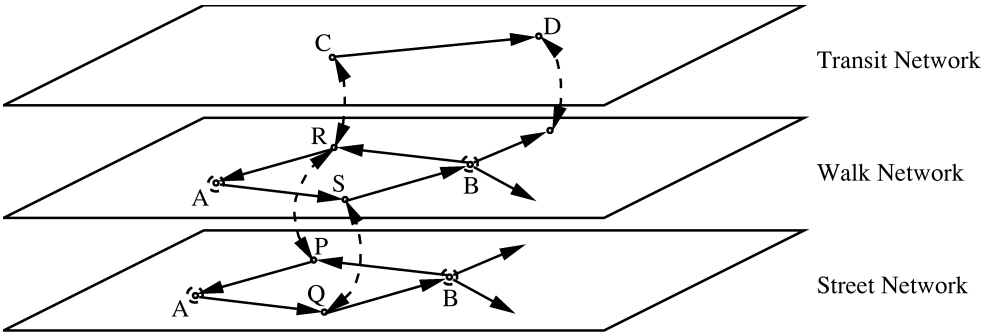

Fig. 4. Conceptual diagram of the Route Planner network, in which parking accessories (P,Q) are in the street layer, activity locations (R,S) in the walk layer, and transit stops (C,D) in the transit layers.

2.9.1 Example

The Activity Generator provides mode preferences for each trip. This information is captured in simple, alphabetical expressions. For example, WCW represents a trip that breaks down as follows:

- W = a walking leg from a traveler's house to his or her car.
- C = a car leg to parking at the place of work.
- W = a walking leg from the parking lot to his/her actual work location.

For the first leg of the trip (the walking leg), the Route Planner searches for possible paths within the walking layer of the network to obtain a walking route from the home to the parking location of the individual's vehicle. After the walking path is found, a series of least-cost driving links is found to obtain a route to a parking location near the work location. A walk route is then developed to move the traveler from the parking lot to the work activity location.

The last two legs of the above route highlight the Route Planner's capabilities. Once the search algorithm is in the car layer, it chooses additional links from the car layer or parks the vehicle and chooses links from the walking layer—whichever is lower in cost. The Route Planner ensures that the final link is a walking link in this example.

Trips that cannot be feasibly planned or that contain questionable legs are marked and provided as output from the Route Planner in the form of the Route Planner anomalous activity file (configuration file key: `ROUTER_PROBLEM_FILE`). These are fed back to the Activity Generator to choose a new activity time or location or mode of travel.

3. ALGORITHM

3.1 High-Level Description

To maintain computational efficiency, the TRANSIMS Network is converted to an internal route network (this is described in Section 3.2 of this chapter). The internal route network represents a weighted, directed graph. The graph's nodes represent intersections and accessory locations (such as parking accessories, activity locations, and transit stops); the arcs (directed edges) represent travel possibilities between node pairs. Internally, all links are unidirectional. Bidirectional TRANSIMS links are represented by two separate links in the Route Planner.

The algorithm underlying the TRANSIMS Route Planner is the classical Dijkstra's algorithm, which finds the shortest paths in a weighted, directed graph. This algorithm can be viewed as a breadth-first search of the graph, starting at the origin node and visiting the other nodes in the order of their (shortest-path) distance from the origin. The actual algorithm used is a direct generalization of Dijkstra's algorithm. In fact, it can be viewed as Dijkstra's algorithm on a larger graph. In full generality, it is described by Barrett, Jacob, and Marathe.[1]

3.2 Route Planner Internal Network

The Route Planner uses information from the TRANSIMS Network and some other files to create the Route Planner Internal Network representation, hereafter referred to as the "internal network". The reason for the internal network is to increase the efficiency of the path-finding algorithm.

3.3 Terminology

1) Node – A physical location in the TRANSIMS Network, such as an intersection, activity location, or bus stop.

2) Link – A street connection from the TRANSIMS Network. Every link has a delay, a layer, and one or more modes of travel associated with it.

3) Edge – A connection between two nodes. Each edge has an associated link and a fraction of the link that it represents.

3.4 Example Transformation

One of the main differences between the TRANSIMS Network and the internal network is that the edges in the internal network are all unidirectional. Any bidirectional links in the TRANSIMS Network are converted to a pair of unidirectional links in the internal network, one in each direction.

[1] C. Barrett, R. Jacob, and M. Marathe: "Models and Efficient Algorithms for Routing Problems in Time-dependent and Labeled Networks," Proc. 6th Scandinavian Workshop on Algorithm Theory, LNCS 1432.

There is a node in the internal network for each node in the TRANSIMS Network, as well as each parking location, activity location, and transit stop.

Each link in the TRANSIMS Network can have accessories attached to it. These accessories represent activity locations, parking, and transit stops, and become additional nodes in the internal network. Transit stops are described in more detail below. Activity locations are placed on the layer specified in the TRANSIMS Network activity location table, while parking locations are always placed on the street layer. For ease of discussion, the following examples assume that all activity locations are placed on the walk layer.

The TRANSIMS Network representation of two nodes with a connecting bidirectional link is shown in Fig. 5. There are six parking locations and five activity locations, connected by process links as shown. One of the parking locations has been designated as a commuter park-and-ride lot.

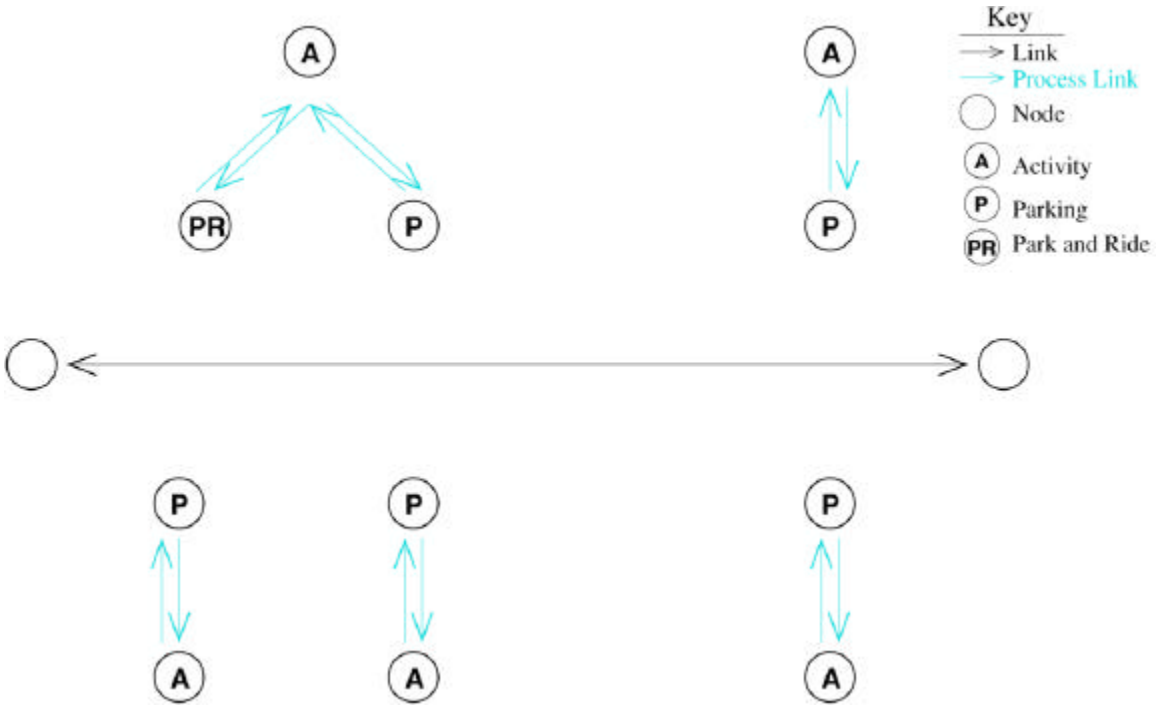

Fig. 5. The TRANSIMS Network representation of two intersection nodes with a connecting bidirectional link.

The corresponding nodes and edges of the internal network representation are shown in Fig. 6. The single link between the two intersection nodes in the TRANSIMS Network has been transformed into four unidirectional links. There is one link in each direction in the street network, as well as a link in each direction in the walk network. If a traveler must park at a park-and-ride lot (i.e., has a mode string of wcwtc), the Route Planner ensures that the traveler passes through the park-and-ride layer when going from the parking locations to the activity location. This is done by internally using the mode string wcpwtw, where p indicates that the traveler must travel on the park and ride layer.

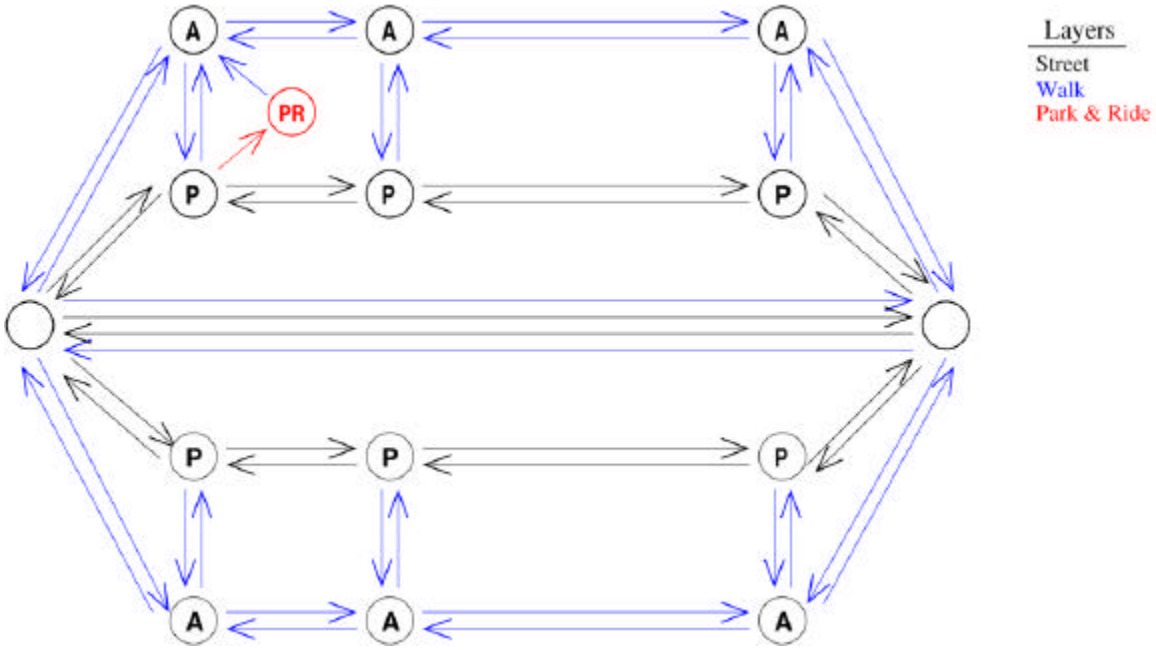

Fig. 6. The corresponding nodes and edges of the Route Planner Internal Network representation.

The edges connecting the two intersection nodes have fraction 1.0. The edges that connect the parking locations are assigned fractions according to the length of the link and the offset of the parking location from the node. The edges connecting the activity locations are similar.

If a link in the TRANSIMS Network does not allow walking, such as a freeway link, any activity locations along that link are still connected by edges in the walk layer. However, no edges are placed between the activity locations and the intersection nodes.

Fig. 7 shows a TRANSIMS Network that is similar to the one shown in Fig. 5, with the exception of a unidirectional link in place of the bidirectional link.

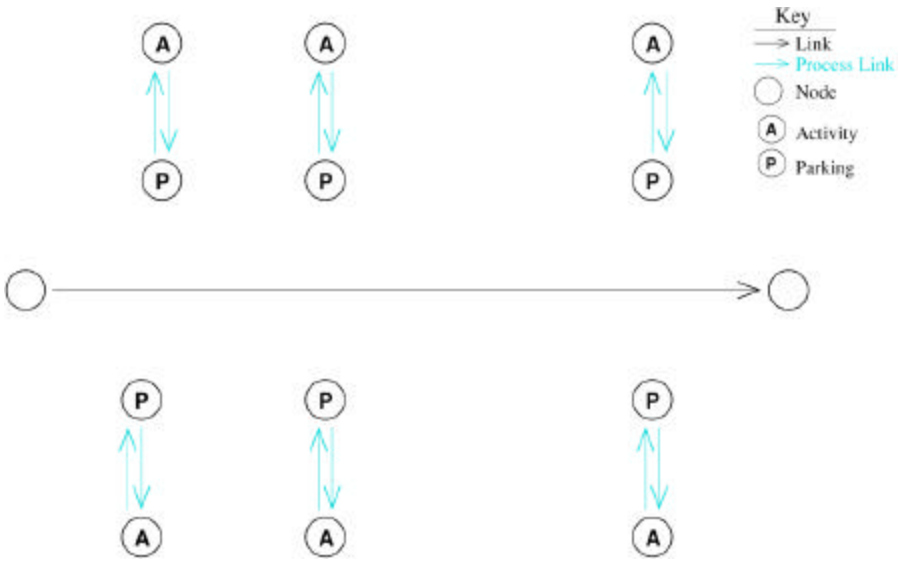

Fig. 7. This TRANSIMS Network is similar to the one shown in Fig. 5, with the exception that this one has a unidirectional link in place of the bidirectional link.

As can be seen from Fig. 8, there are edges in one direction only on the street layer. However, there are still edges in both directions on the walk layer. This is because walking can always be performed in either direction.

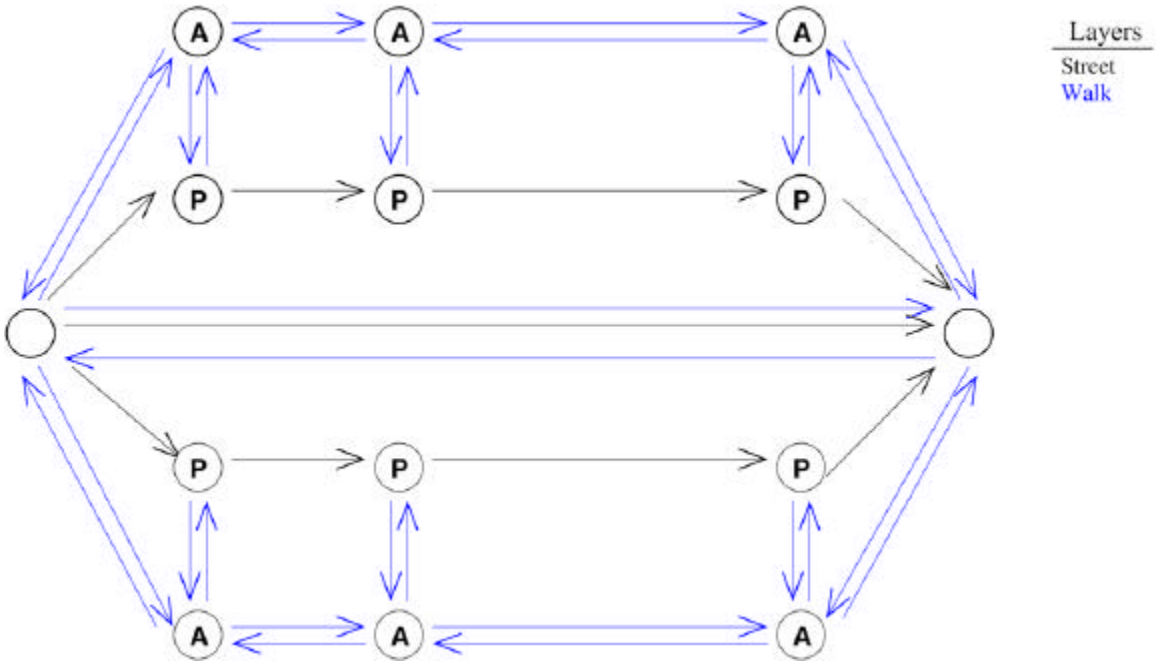

Fig. 8. This figure shows that there are edges in one direction only on the street layer.

3.5 Network Assumptions made by the Route Planner

- No location (parking or activity) is located off the end of its link. Their locations on a link in TRANSIMS are specified by the distance from the endnode of the link (the value called “offset”). The Route Planner assumes that no offset is negative, and that every offset is less than the length of the corresponding link. If this assumption is not satisfied, the Route Planner prints warnings. It proceeds in planning, but its behavior (especially with respect to calculating distances) is not defined.
- No two parking locations lie on the same link and have the same offsets.
- Each activity location is adjacent to a parking location. (This is not important if no trips are planned from the activity location that starts with `WC` or to the activity that ends with `CW`). This restriction will be removed in a future version of the Route Planner.

3.6 Transit

Information about the transit system comes from the TRANSIMS Network transit stop table (configuration file key: `NET_TRANSIT_STOP_TABLE`), the transit route file (configuration file key: `TRANSIT_ROUTE_FILE`), and the transit schedule file (configuration file key: `TRANSIT_SCHEDULE_FILE`).

Each transit stop in the transit stop table is represented by a node in the transit layer for each type of transit that serves that stop. Each route in the route file has its own layer, containing a node for each stop on the route called route nodes. There are process links connecting each transit stop to the corresponding route nodes. The route nodes are connected by links, in the order that the route nodes appear in the route file. The stops in a particular route must be unique.

Each transit stop must be explicitly connected to the walk network with process links to appropriate activity locations. The delays for the route links are taken from the route schedule file. The delays for these links are represented by a piecewise constant delay function.

Fig. 9 shows the TRANSIMS Network representation of two streets with bus stops and two bus routes connecting them.

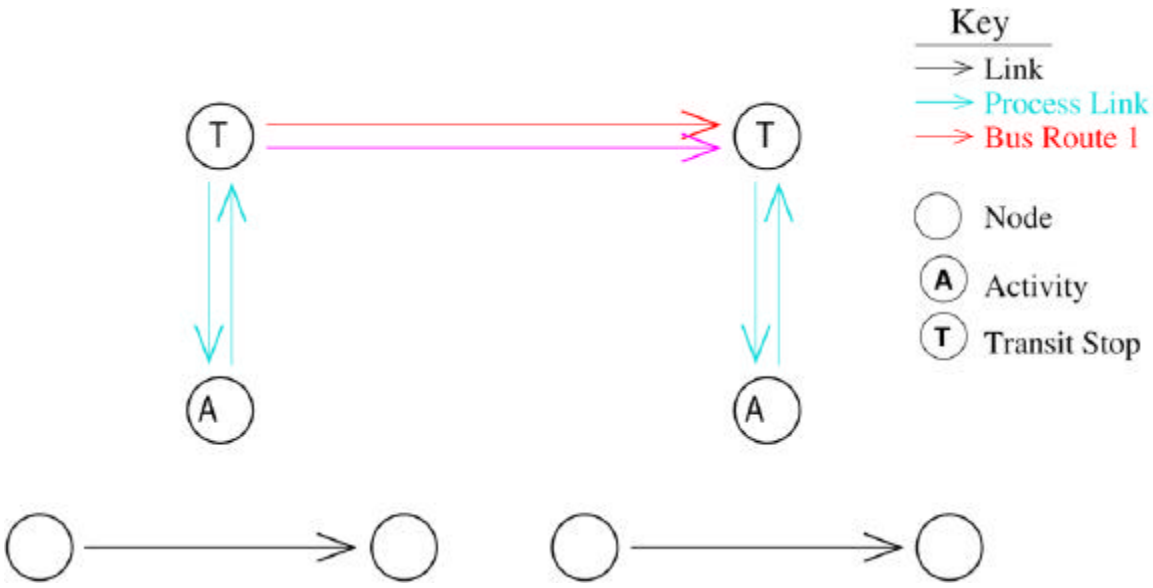

Fig. 9. TRANSIMS Network representation of two bus routes.

Fig. 10 shows the corresponding Route Planner Internal Network representation. Note that there are five different layers in the internal network:

- the street layer containing the intersection nodes,
- the walk layer containing the activity locations,
- the bus layer containing the bus stops, and
- a layer for each bus route.

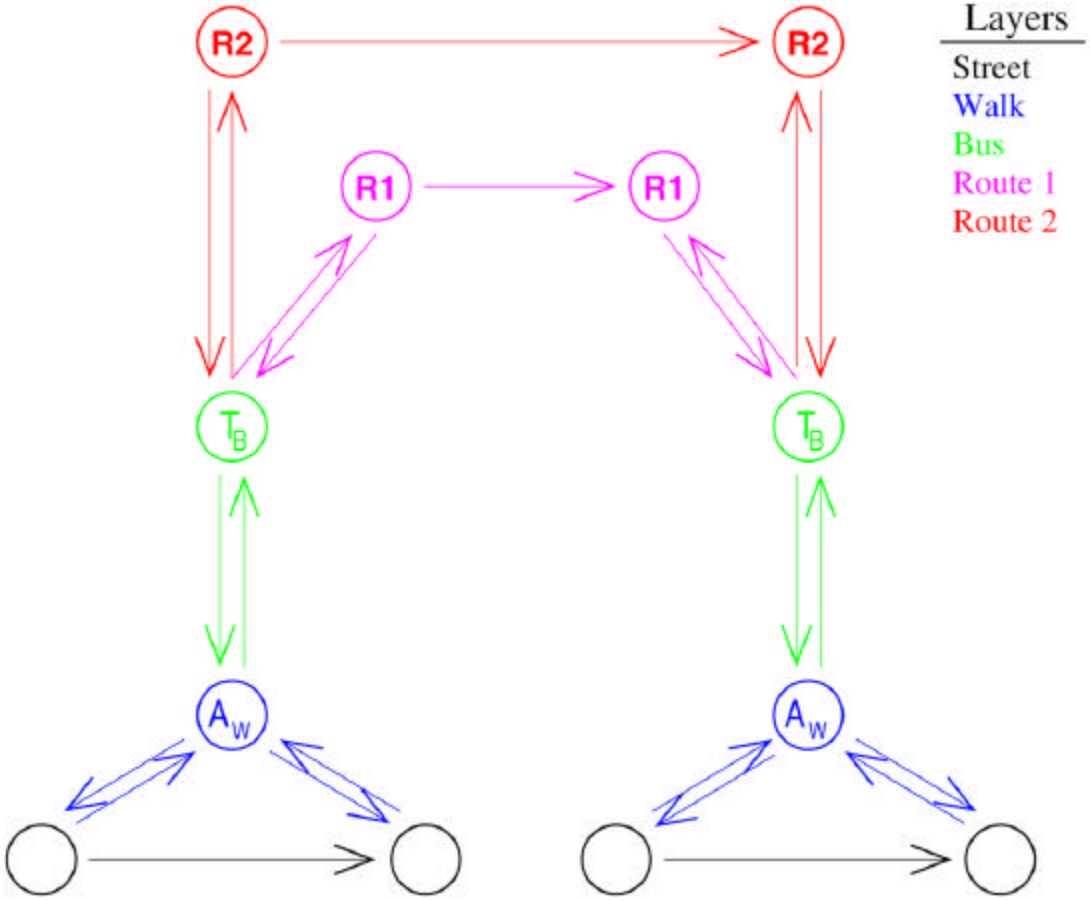

Fig. 10. Route Planner Internal Network representation corresponding to Fig. 9.

Fig. 11 and Fig. 12 show a more complex example with two bus routes and a light rail line. Note that there is only one transit stop for both bus and light rail in the TRANSIMS Network, but separate stops for different types of transit in the internal network.

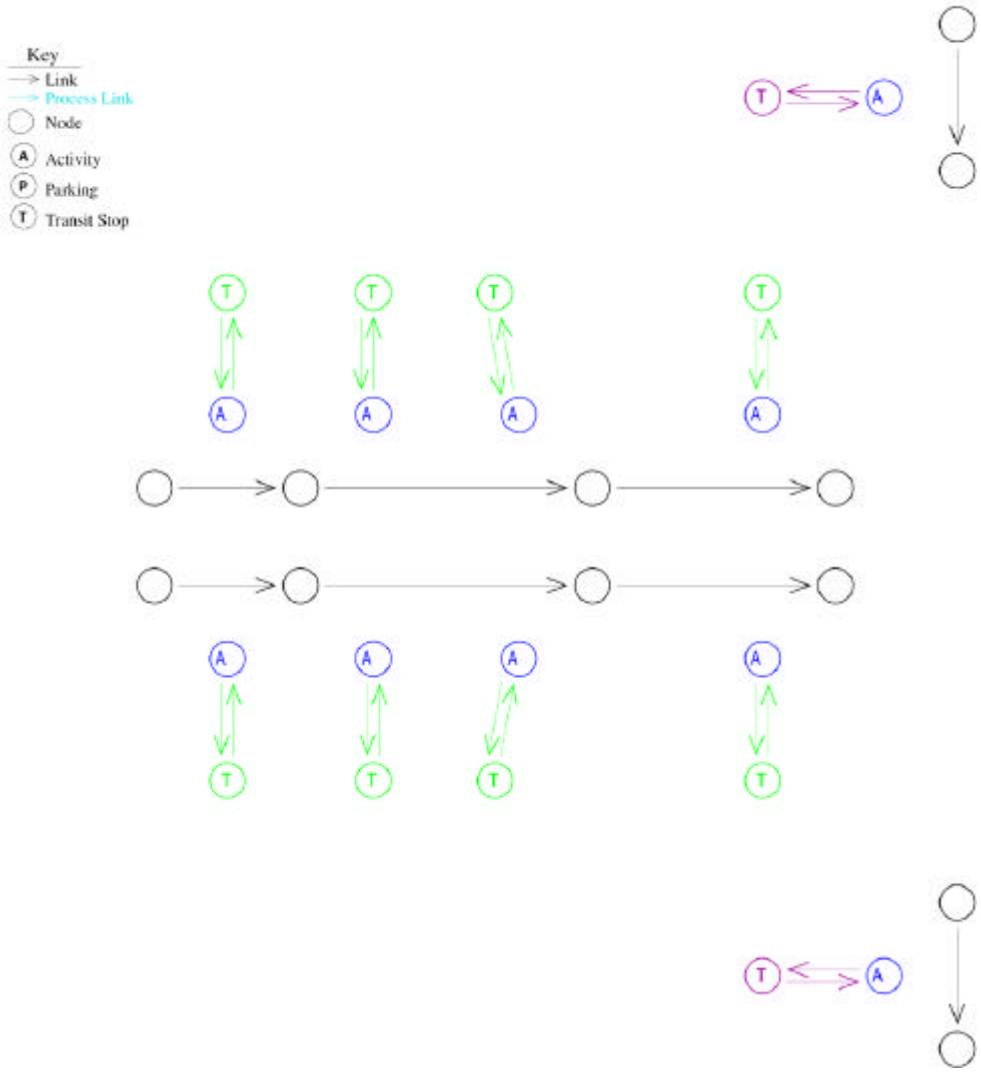

Fig. 11. TRANSIMS Network representation of a complex transit network with two bus routes and a light rail line.

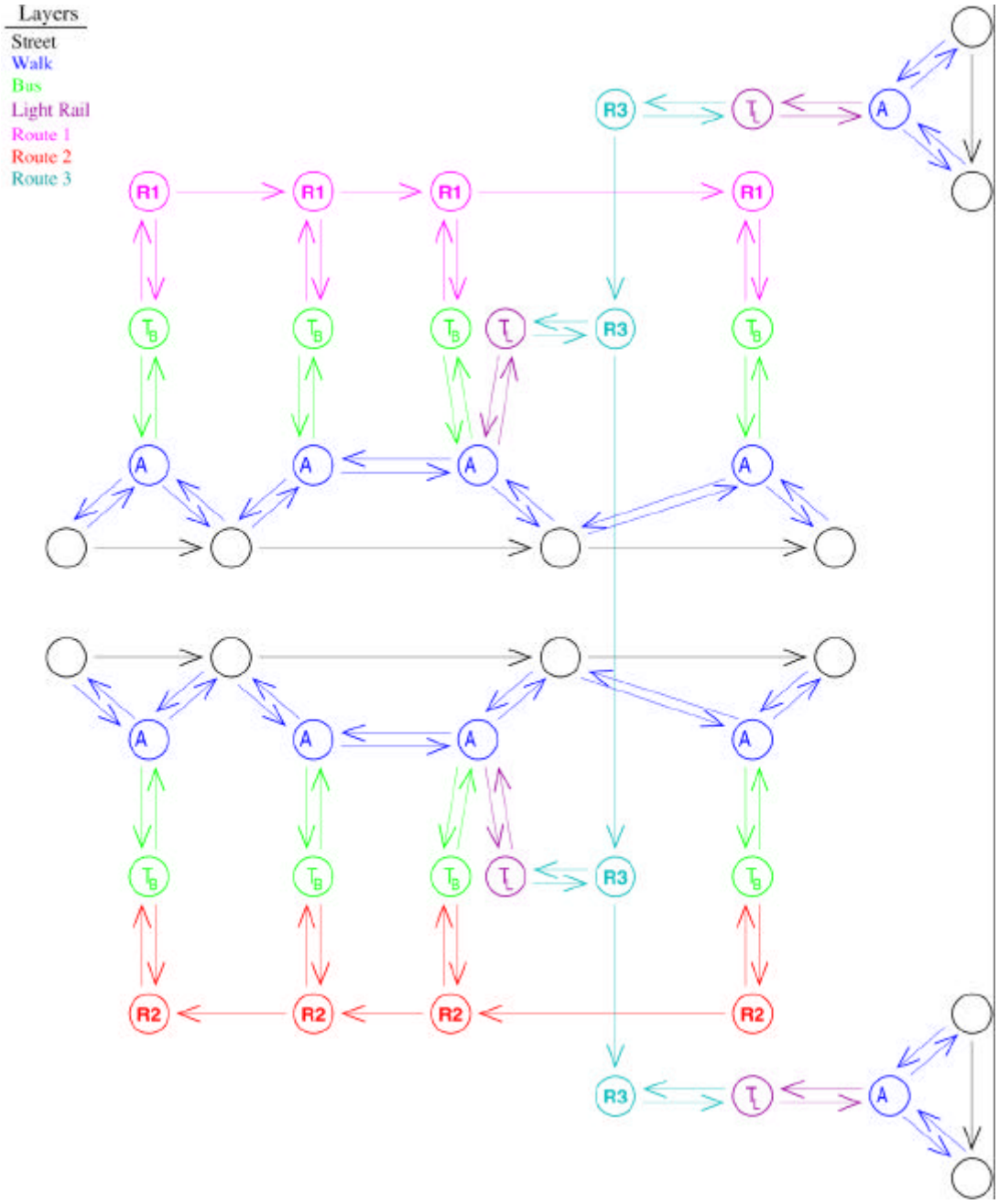

Fig. 12. Route Planner Internal Network representation corresponding to Fig. 11.

3.7 Cost

There are several ways to determine the “cost” of a trip. The Route Planner uses travel time to determine the shortest path through the transportation network. It also computes monetary cost and distance. The Generalized Cost Function is not yet supported.

3.7.1 Travel Time

Each link has a delay associated with it. Links on the street layer have a delay for driving on that link. Links on the walk layer have a delay for walking on that link. Transit links have a delay for the time arriving at a transit stop and the time at which the transit vehicle may be exited at the following stop. This delay takes into account the time spent waiting for the transit vehicle to arrive, based on its schedule. Delays can either be constant, such as walking delays, or dependant on the time of day.

The default delay for a street link is the free speed delay. It is calculated from the free speed on that link and the length of the link. The actual delays calculated by the Traffic Microsimulator are used to provide more accurate information. These delays are given in the link delay file (configuration file key: `ROUTER_LINK_DELAY`). Each delay represents the average delay experienced for the vehicles that traversed the link, averaged over a 15-minute interval.

The delay for walking or biking on a link is determined from the length of the link and the walking speed (configuration file key: `ROUTER_WALKING_SPEED`) or biking speed (configuration file key: `ROUTER_BIKING_SPEED`). There are also delays for entering transit vehicles (configuration file key: `ROUTER_GET_ON_TRANSIT_DELAY`) and exiting transit vehicles (configuration file key: `ROUTER_GET_OFF_TRANSIT_DELAY`). The transit delays are used to keep travelers from changing transit vehicles to save a few seconds of travel time.

Process links can also have a delay associated with them. For example, the delay involved in parking a vehicle in a lot can be represented by the delay on the process link from the parking location to any adjacent activity locations.

To increase the effectiveness of Traffic Microsimulator/Route Planner feedback, noise can be added to the link delays. The maximum amount of noise to add to the link as percentage of the link delay can be specified (configuration file key: `ROUTER_NOISE_DELAY`). If the delay for a link is d and the specified noise percentage value is n, the reported delay will be in the interval ($d-nd$, $d+nd$). Fractional links that are used to access parking accessories always have the maximum amount of noise added to them. This is to ensure that traveling on the partial links is always at least as expensive as traveling on the associated full link.

3.7.1.1 Heuristics

To increase performance, the links that the Route Planner examines can be reduced. This is done by artificially increasing the delay for links that lead in the wrong direction. For example, assume that the source location for a trip is in the southern part of the network and that the destination location is directly north. Links that head north will be preferred over links that lead east or west. The farther from north that a link leads, the less likely it is that the link will be considered.

The *Sedgewick-Vitter heuristic* can be used for Euclidean graphs. The heuristic allows finding almost optimal shortest paths between nodes in a Euclidean graph. A parameter

called `overdo` (configuration file key: `ROUTER_OVERDO`) allows for a tradeoff between the running time and optimality of the paths found. The internal network is not strictly Euclidean, since only certain nodes may be reached from each node (the graph is not complete), but we have found that the paths produced with moderate values, such as `overdo = .25`, look quite realistic and bring a considerable improvement to running time.

However, if this heuristic is used, the plans will be less sensitive to feedback (i.e., changing the link delays). The larger the value of `overdo`, the longer congestion will be tolerated by the Route Planner before alternative routes are taken.

In addition, with `overdo` turned on, certain geometric configurations in the network will cause the Route Planner to prefer low-speed links that head in the correct direction over high-speed links that head in an incorrect direction. For example, the Route Planner may create a plan that causes a traveler to exit a freeway via a ramp, only to reenter several links later, rather than remaining on the freeway.

If the value of overdo is 0, and the delay noise is 0, then the optimal (i.e., least cost) path will be found for the particular mode string used.

3.7.2 Distance

For each route, the distance traveled by traversing the route is calculated. The distance for a transit leg is the sum of the Euclidean distances between each pair of transit stops. For auto, walk, and bike legs, the distance is the sum of the length of the links traveled. For magic move legs, the distance is the Euclidean distance between the source and destination activity locations.

3.7.2.1 Monetary Cost

In addition to travel time delay, process links can also have an associated monetary cost. This can be used to account for parking fees, transit fares, and tolls. All costs are expressed as cents.

The cost of parking is represented by the cost on the process links from the parking accessory to any connected activity locations in the Process Link Table.

There are two types of transit costs, referred to here as *fixed fare* and *variable fare*. Fixed fare means that the fare is calculated based on where the transit vehicle is entered, regardless of where it is exited. A variable fare depends on where the transit variable is entered and exited.

A fixed fare is handled similarly to parking costs. The price of the fare is the process link cost from activity location to transit stop in the Process Link Table.

A variable fare is handled by transit fare zones (TFZ). Each transit stop is assigned a TFZ. The transit fare table contains the cost of traveling between each pair of TFZs by transit type. More information can be found in Volume Two (*Networks and Vehicles*) Section 2.1.3.

Any individual links that have a cost associated with them (e.g., tolls) can be listed in the link cost file (configuration file key `ROUTER_LINK_COST_FILE`). This file contains pairs of link ID and cost.

3.7.2.2 Generalized Cost Function

To more accurately model mode choice, the concept of a generalized cost function (GCF) has been developed. The GCF allows other factors in addition to travel time and monetary cost, to be taken into account when determining a plan for a traveler. These other factors are included in the "cost" of a trip. The importance of the monetary cost of a trip may be modified depending on a traveler's income. A greater penalty for traveling on congested links can be imposed by calculating the difference between actual delay and free speed delay. Transit transfers may impose a higher cost than the actual delay involved. The GCF currently reported is simply the travel distance.

3.7.3 Current Limitations

There are several limitations to the way the cost and distance are currently computed. These may be fixed in a future version. Fixed transit costs and transit distances are all combined in the first transit leg if multiple routes are used in one trip. For example, the trip in Table 13 will be reported as in Table 14.

Table 13. Actual trip.

Leg Mode	Distance	Monetary Cost
w	0.5 km	0
t – Bus Route 1	2.0 km	100
w	0.1 km	0
t – Bus Route 2	1.5 km	150
w	0.1 km	0

Table 14. Reported trip.

Leg Mode	Distance	Monetary Cost
w	4.2 km	250
t – Bus Route 1	0 km	0
w	0 km	0
t – Bus Route 2	0 km	0
w	0 km	0

Similarly, distance and parking costs for the walk leg from the parking location to the activity location are included in the auto leg of the trip.

4. ROUTE PLANNER RUNTIME CONFIGURATION

4.1 Logging Configuration File Keys

The amount of information output by the Route Planner can be controlled in several ways. The logging configuration file keys `LOG_ROUTING`, `LOG_ROUTING_DETAIL`, and `LOG_ROUTING_PROBLEM` control the amount of logging information generated. Logging information is normally sent to standard output. The configuration file key `ROUTER_LOG_FILE` can be used to direct the logging output to a specific file.

`LOG_ROUTING` generates information about the general progress of the Route Planner. This can normally be turned on (set to 1 in the configuration file). `LOG_ROUTING_DETAIL` generates copious amounts of logging on information and is normally turned off for normal execution. `LOG_ROUTING_PROBLEM` duplicates the information in the Route Planner anomalous activity file.

If the configuration file key `ROUTE_DISPLAY_PATHS` is set to 1, the Route Planner will generate the specific nodes traversed for each path found, even for unsimulated modes such as walk. Setting this configuration file key will generate large amounts of output.

4.2 Other Configuration File Keys

There are several other configuration file keys that can affect the execution of the Route Planner. The configuration file key `ROUTER_SEED` allows the seed of the random number generator to be set.

If the configuration file key `ROUTER_COMPLETED_HOUSEHOLD_FILE` is set, household IDs will be written to this file as they are completed. This allows restart capability, as any households whose IDs are in this file need not be replanned.

The configuration file key `ROUTER_INTERNAL_PLAN_SIZE` controls the size of data structure used to store plans before they are written to the plan file. The size should be larger than the largest possible number of nodes used in a path through the network. A size of 1000 is sufficient for small networks, while 4000 may be needed for a large network.

Appendix C provides a complete list of the Route Planner configuration file keys.

5. PLAN RETIME

The program *RetimePlans* has the ability to change the duration of existing plans due to updated link delay times or transit schedule files. No attempt at ensuring the validity of retimed plans is made, only the duration of the plans is changed. Existing plans are read from the plan file (configuration file key `PLAN_FILE`), and the duration of each selected path is recalculated. The new plans are written to the retimed plan file (configuration file key `ROUTER_RETIME_PLAN_FILE`). If the retime traveler file (configuration file key `ROUTER_RETIME_TRAVELER_FILE`) exists, only plans for travelers whose IDs are in this file will be retimed and written to the retimed plan file.

If the configuration file key `ROUTER_RETIME_MODES` is specified, only plans with the given modes will be retimed. Currently, only retiming of auto plans is supported.

6. ROUTE PLANNER UTILITY PROGRAMS

6.1 *MakeHouseholdFile* Utility

The *MakeHouseholdFile* utility allows the creation of a set of household files that collectively contain all of the households contained in the population file (configuration file key `ACT_POPULATION_FILE`). The program is executed as

```
MakeHouseholdFile <configuration file> <number>
```

where `number` is the number of household files to create. The household files are named according to the configuration file key `ROUTER_HOUSEHOLD_FILE` in the configuration file, with the number of the processor appended as *.txx*, where *xx* is the rank expressed as a two-digit base-26 number (i.e., the sequence is AA, AB, .. AZ, BA, BB, …).

6.2 *10to26* and *26to10* Utilities

The utilities *10to26* and *26to10* are simple utilities for converting between base-10 and base-26.

```
10to26 <integer>
```

will output the base-26 representation of integer.

```
26to10 <XX>
```

will output the base-10 representation of the base-26 number ‘XX’

6.3 *CatIndices* Utility

The *CatIndices* utility provides merging of TRANSIMS route plan indices to produce a combined and sorted index. *CatIndices* has been optimized to concatenate and resort indices much faster than the alternative utility, *PlanFilter*. Plan files are indexed by traveler (*.trv.idx*) or by time (*.tim.idx*). *CatIndices* creates either the traveler index using an existing time index or a time sorted index using a traveler index. It is assumed that either the time or traveler sorted index, as appropriate, exists and is up-to-date for each of the input plan file arguments. The default is to create a time-sorted index from an existing traveler index(es).

Format:

```
% $TRANSIMS_HOME/bin/CatIndices [-h] [-f] <outIndexName> <PlanFile> [<PlanFile>...]
```

where

`-h` Prints a description of the program and the allowed arguments

`-f` creates a travel-sorted index from an existing time-sorted index(es)

Example:

```
% $TRANSIMS_HOME/bin/CatIndices plans.all plans.all.[0-9]*[0-9]
% $TRANSIMS_HOME/bin/CatIndices -f plans.all plans.all.[0-9]*[0-9]
```

The first command creates the time-sorted index "*plans.all.tim.idx*" from existing traveler-sorted indexes "plans.all.[0-9]*[0-9]. The second command creates the traveler-sorted index "*plans.all.trv.idx*" from existing time-sorted-indexes "plans.all.[0-9]*[0-9]".

6.4 *PlanFilter* Utility

The *PlanFilter* utility provides sorting. merging, selection, and validation of plans. It constructs two indexes for each input and output plan file it touches—one sorted by time, and the other by traveler. Currently, existing indexes are used if they are up-to-date. All times are measured in seconds since midnight.

If the `-v` option is used, only valid plan sequences are included in the output indexes, and a brief description of errors encountered in each plan is written to standard output. Use of this option is recommended before using any plan file in the Traffic Microsimulator because it can detect many errors that are likely to cause the simulator to crash. *PlanFilter* can detect the following conditions in any plan:

- Trip and leg ID sequence errors; that is, IDs out of order or not consecutive—flagged as "bad trip id" and "bad leg id" respectively.
- Leg not starting from the previous leg's destination or starting locations not found in the network tables—flagged as "bad start accessory".
- Destination not found in the network tables—flagged as "bad end accessory".
- Leg's departure time earlier than previous leg's estimated arrival time—flagged as "bad activation time".
- Zero or negative duration—flagged as "bad duration".
- Estimated arrival time earlier than departure time—flagged as "bad stop time".

If the leg requires driving, *PlanFilter* will also detect the following conditions:

- Links or nodes not found in the network tables or not contiguous—flagged as "bad route".
- Driver moving from one link to another with no lane connectivity between them—flagged as "no allowed lane for turn".
- Driver moving from any link onto the same link—flagged as "plan requires U-turn".

If the `VEHICLE_FILE` configuration file key is set in the configuration file specified with the `-v` command line argument, *PlanFilter* can use information in the Lane Use table and the vehicle restriction field of the Link table. In this case, it detects the following conditions:

- Driver is using a non-existent vehicle—flagged as "vehicle not found in vehicle file".
- Driver attempts to drive down a link which does not allow the type of vehicle being used—flagged as "violates vehicle restriction on link".
- Vehicle is not allowed in lane required for moving to the next link. (For example, a car using a bus-only left turn lane). This is also flagged as "no allowed lane for turn".

Not all of these conditions are considered serious enough to invalidate the plan and prevent it from being included in the output indexes. In particular, U-turns are not prohibited (unless there is no lane connectivity allowing a U-turn at the desired node) and missing vehicles are not considered a problem. Processing is discontinued for each leg until a serious error is encountered. All plan legs up to the first invalid leg for a traveler are included in the output.

Usage:

```
PlanFilter [-h] [-d] [-f] [-w] [-v netConfigFile] [-s startTime] [-e endTime] [-t
travId]* [-r <travFile>] [-o <outFile>] <planFile>*
```

where:

h = print this message
d = defragment the file: create a new plan file containing the merged, filtered plans; the *-o* flag must accompany this flag
f = sort output by traveler
v = validate each trip chain:
netConfigFile must be a TRANSIMS configuration file specifying a network database (Validation may be time-consuming.)
s = include only legs whose (estimated) departure time is >= startTime
e = include only legs whose (estimated) arrival time is <= endTime
t = include only legs for traveler travId; implies the *-f* flag
(May appear an arbitrary number of times.)
r = include only legs for travelers specified in travFile; implies the *-f* option
(May appear together with the *-t* options.)
o = place output in outFile; default is standard output

Arguments that do not start with "-" are assumed to be input plan files.

6.5 *DistributePlan* Utility

The purpose of the *DistributePlan* utility is to create a separate pair of indexes into a plan file for each processor in a multiprocessor run of the microsimulation. Each leg of a trip is assigned to the processor that has responsibility for the starting accessory of that leg. This allows the processors to get travelers into the simulation more efficiently than if each processor had to read in every leg, discarding those that it did not need.

DistributePlans uses a mapping from accessory type and ID to CPU number. This mapping, or partition, is created during a simulation run as specified by the values of certain configuration file keys. It is saved in a file specified by the

PAR_PARTITION_FILE configuration file key if the PAR_SAVE_PARTITION configuration file key is set. Note that, if run time information is saved during the simulation (using the PAR_RTM_INPUT_FILE) and that information is used to partition the network on the next run (by setting the CA_USE_RTM_FEEDBACK configuration file key), the partition can change from one run to the next.

DistributePlans can also generate the partition if none is present. In this case, the partition can be saved and used by the microsimulation (by turning off both the PAR_USE_METIS_PARTITION and PAR_USE_OB_PARTITION configuration file keys).

DistributePlans creates an index file for each processor in the partition, using a simple naming convention that allows the individual slaves to find the correct index file if it exists.

For each leg in a plan file specified by the PLAN_FILE configuration file key, *DistributePlans* determines the starting location's accessory type and ID. Next, it finds the processor number assigned responsibility for that location. Finally, it places an index entry for the leg in the file for that processor. The underlying data is not moved.

There is one additional task handled by *DistributePlans*. When a trip's legs are distributed, it becomes difficult for any processor to know whether a particular leg represents the first or last leg a traveler will undertake during the course of the simulation. This information is required because on a traveler's first leg, the associated object must be created within the simulation. On all other legs, the traveler object must not be created—instead the simulation must wait for the traveler object to arrive at that leg's starting location before allowing it to continue. Similarly, but not quite as importantly, efficient use of memory requires deleting the traveler object at the end of it's last leg.

DistributePlans ensures that the appropriate information about each traveler is made available to the simulation. It places an index entry for the first leg of each traveler's trip into each distributed index. This, in combination with the ability of the microsimulation to use both a traveler ID sorted index and a time sorted index allows it to correctly create and destroy travelers.

Usage:

```
DistributePlans <config-file>
```

6.5.1 *DistributePlan* Configuration File Keys

The configuration file keys listed in Table 15 are used when a partition already exists.

Table 15. Configuration file keys if a partition exists.

Configuration Key	Description
PAR_PARTITION_FILE	Name of a file providing a mapping from nodes to processors. This file also includes node coordinates, so it can be used to display the partition.
PLAN_FILE	The name of a plan file to distribute over the partition.
NET_*	The configuration file should also contain all the NET_ configuration file keys.

The configuration file keys listed in Table 16 are used to generate a partition if one does not already exist.

Table 16. Configuration file keys to generate a partition.

Configuration Key	Description
PARTITIONER_USE_NETWORK_CACHE	If set, the code will read in a binary cached version of the network.
GBL_CELL_LENGTH	The length of a CA cell in meters.
PAR_MIN_CELLS_TO_SPLIT	Splitting short links can cause problems in the dynamics of the microsimulation. No links with fewer cells than this will be split.
PAR_SLAVES	The number of processors in the partition.
PAR_RTM_PENALTY_FACTOR, PAR_RTM_INPUT_FILE, CA_USE_RTM_FEEDBACK	See the description in the software modules volume, Microsimulation section on configuration file keys.
PAR_HOST_COUNT, PAR_HOST_CPUS_<n>, PAR_HOST_SPEED_<n>	These parameters are used to describe the machine environment. Relative processor speed will be taken into account when creating the partition.
PAR_USE_METIS_PARTITION, PAR_USE_OB_PARTITION	If PAR_USE_METIS_PARTITION is set, the partition will be determined using the METIS graph partitioning library. If PAR_USE_OB_PARTITION an is set, orthogonal bisection algorithm will be used. If neither is set, the partition specified in the PAR_PARTITION_FILE will be used.
PAR_SAVE_PARTITION	The partition will be saved in PAR_PARTITION_FILE only if this is set.

6.5.2 Troubleshooting

If a very large number of processors are used, the algorithm may run into an operating system limit on the number of open file descriptors allowed.

Distributing the indexes makes the plan-reading phase of the microsimulation more efficient. However, there may be I/O considerations that are important when a large number of processors are trying to gain access to the same underlying data files. This

problem could be addressed by using the *PlanFilter* tool to create a separate data file for each of the indexes created and the *IndexPlanFile* tool to recreate the indexes, now pointing at the distributed plan files instead of a global file.

6.6 *CongestedLinks* Utility

The *CongestedLinks* utility counts how many drivers intend to be on each link within a specified time window. Its input is a plan file. The demand estimate it provides does not take into account interactions among vehicles or capacity constraints and jam formation. The output reports both raw counts and a count normalized by the number of lanes and length of the time window, which is the effective maximum flow rate of each link in the CA. The output can be fed directly into the output Visualizer as a Link Data format file.

Usage:

```
CongestedLinks [-h ] [-i <time_inc>] [-t <threshold>] [-s <start_time>] [-e
<end_time>] <configFile> <planFile> <outFile>
```

All times are in seconds since midnight

The following are options:

`-i` specifies a time increment. Default = 900

`-t` specifies a threshold - only links whose density is over capacity by <threshold> are included in the output

`-h` gives a help message

`-s` only vehicles on links after this time are counted. Default = 0

`-e` only vehicles on links before this time are counted. Default = 86400

The configuration file, plan file and log file names are all required, and must appear in the order shown. Note that the configuration file should contain all `NET` configuration file keys, and all `ROUTER` configuration file keys. The plans, however, are taken from the specified plan file instead of the one specified by the `PLAN_FILE` configuration file key.

Example:

Here is sample output from *CongestedLinks*:

```
TIME  LINK    NODE   LANE  norm_flow  COUNT  SUM       to_node
8550  112192  46802  -1    0.0165278  14     104.156   46281
8550  112363  47203  -1    0.0165278  14      78.4505  46802
8550  112527  47703  -1    0.0153472  13      96.6039  47203
8550  112675  48180  -1    0.0177083  15     106.859   47703
8550  112777  48431  -1    0.0165278  14      60.6613  48180
8550  113482  49908  -1    0.0177083  15     144.165   49289
8550  114099  50911  -1    0.0100347  17     134.467   50348
8550  118278  57439  -1    0.0109091   9      24.2342  57468
8550  118283  57472  -1    0.0109091   9      24.317   57439
```

The `SUM` column is arbitrary and experimental; it can be safely ignored.

6.7 *RearrangePlans* Utility

Before the Traffic Microsimulator can execute the plans produced by the Route Planner, some manipulation of the plan files is required. The Route Planner naturally distributes computation across CPUs by household, producing approximately 50 individual plan files, each containing plans for a different set of households. The Traffic Microsimulator distributes computation across CPUs geographically and executes plans in time order for the most part. While it is technically possible to use the plan files created by the Route Planner directly in the Traffic Microsimulator, it is extremely inefficient because the Microsimulation would be forced to open and close files and position them correctly for reading each and every plan.

The necessary file manipulation has been automated in the script *RearrangePlans*. The script is specific to the Linux cluster on which we are running, but can be tailored to other architectures. This section gives a step-by-step description of the operation of this script and other new utilities it requires, and describes why it is structured as it is.

The first step is to create indexes for each of the 50 or so plan files created by the Route Planner, the transit driver plans, the truck driver plans, and the itinerant plans (which are themselves split into separate files for the a.m., p.m., and mid-day peaks, and all the rest). The indexes will be required in the next step. Since each of the indexes is completely independent of the others, we can use a separate node for each one. (Although the cluster we work on has two CPUs per node, it is usually more convenient to run only one job per node). We invoke the executable *IndexPlanFile* for each plan file on a separate node and wait for all of them to finish. The Route Planner creates plan files with a base name plus the extension *.txx* where the *x*s are replaced with capital letters starting from *AA*, *AB*, and continuing through the alphabet (e.g., *<base>.tAA*, *<base>.tAB*, etc. *IndexPlanFile* will create the indexes *<base>.tAA.tim.idx* and *<base>.tAA.trv.idx* for the first plan file, and similarly for the others.

The next step is to create plan files incorporating all of the population, transit, truck, and itinerant plans in time-sorted order. These are the data files that will be used by the Traffic Microsimulator. It is not crucial that all of the data be in a single file, only that it be ordered by expected departure time. This part of the computation can be distributed by allowing each CPU to consider only those plans whose expected departure time is within a certain interval. Because we have enough nodes available and it is convenient for other analyses, we have chosen to split the plan files into half hour pieces. Thus, each of 48 CPUs runs *PlanFilter* on all of the input plans, extracting only those whose departure times are within a half-hour window. This fixed-time window does not result in an ideal partition of the work, since many more trips start during peak travel times than, say, 4:30 - 5 a.m.

The plans for each half hour are placed in a file labeled by the end time of that window (in seconds since midnight). For example, plans starting in the interval (7:00, 7:30] can be found in *<base>.27000*. The corresponding indices *<base>.27000.tim.idx* and *<base>,27000.trv.idx* are also created. The command that accomplishes this is:

```
PlanFilter -d -o <base>.27000 -s 25201 -e 27000 <base>.tAA <base>.tAB ...
```

The `-d` argument causes a data file containing the plans to be created in addition to the indices. Logging output from this command is placed in the file *log.27000* in the directory where the population plans reside.

The Traffic Microsimulator expects a single plan file name. As discussed in the indexing section, only indexes for that plan file need exist. In this step, we create two indexes that point to the 48 time-sorted, half-hour interval plan files created above. This process cannot be distributed, since we are creating a single index from a set of them. However, we will need to create two indexes (sorted by time and traveler), so we can do them simultaneously on two CPUs. At our site, two of the cluster's nodes have large local temporary disks. It is much more efficient to create the indexes on a local disk than across NFS, so we use those two nodes. This step uses the utility *CatIndices*, which has been optimized to concatenate and resort indexes much faster than the alternative, *PlanFilter*. A command line argument specifies which of the two indexes (traveler or time) it is to build, and other arguments where the result should go, and what the input plan files are. The two commands are:

```
CatIndices plans.all plans.all.[0-9]*[0-9]
CatIndices -f plans.all plans.all.[0-9]*[0-9]
```

The first creates the time sorted index *plans.all.tim.idx*; the second creates the traveler sorted index *plans.all.trv.idx*.

The final step needed to prepare plans for the Traffic Microsimulator is geographic distribution of the time-sorted plans. Each CPU in the Traffic Microsimulator is responsible for a different geographic area of the network. It only needs to read in plans that start within that area. Other plans will be passed to the CPU in messages from other CPUs as needed. If there are N CPUs in use, failing to distribute the plan files geographically will cause each CPU to read roughly N times as many plans at it needs to, slowing down the Traffic Microsimulator. Fortunately, we need not distribute the plan data itself to each CPU. All that is needed is a CPU-specific index containing only the plans that start on that CPU. These indexes are created in parallel (one process for each CPU) using the executable *DistributePlans*.

After this step has been taken, the Traffic Microsimulator can be (and is) run using the script *RunCA*. Sometimes it is also useful to run the Collator at this point, or the *CongestedLinks* program to estimate demand as a function of time on each link from the plan files.

7. PLAN FILES

7.1 Overview

This section gives the protocol of the TRANSIMS plan file interface between the Route Planner and the Traffic Microsimulator.

7.2 File Format

The TRANSIMS code supplies a library of C routines, as well as a TPlan C++ object that can read and write this format.

The format consists of a required "header" and a set of "mode-dependent data." The header contains information common to every kind of leg. Code that uses the plans may choose to ignore some or all of the mode-dependent data. For example, the Traffic Microsimulator will not simulate walking or bicycling, but it will use the estimated duration from the Route Planner.

Because the origin, destination, and expected duration of any leg are available in the header information, the simulation does not require any data in the mode-dependent part of a walk leg.

7.2.1 Data Definitions and Format

A plan file contains a series of records, each of which specifies a single leg of a traveler's trip. Each record contains the fields shown in the table found in Appendix A, in the order shown, separated by white space [space, tab, and/or a single newline]. The field names are not written in the data file. There is a blank line separating each pair of records. The file is written in ASCII text. Efficiency concerns are addressed by accessing plan files through an index. See the *Index* section for details.

The combination of `duration`, `stop time`, and `max time` allows flexible specification of departure times. For example, attending a movie might be encoded as follows:

```
duration = 0 seconds;
stop time = 20*3600 + 30*60 = 73800;
maxTime = true;
```

which means, "this activity ends at 8:30 p.m., or as soon as the traveler arrives, whichever is later."

Similarly, work might be encoded as follows:

```
duration = 8 hours;
stop time = 17*3600 = 61200;
maxTime = true;
```

which means “stay at work until 5:00 p.m., or eight hours after arrival, whichever is later.”

Shopping at lunch might be encoded as follows:

```
duration = 0.5 hours;
stop time = 12*3600 + 45*60 = 45900;
maxTime = false;
```

which means “shop for half an hour or until 12:45 p.m., whichever is earlier.”

7.2.2 Mode-Dependent Data

Mode-dependent data are written by the Route Planner and interpreted by the Traffic Microsimulator. Appendix B provides such data for review.

7.3 Plan Library Files

Table 17 records plan library files.

Table 17. Plan library files.

Type	File Name	Description
Binary Files	*libTIO.a*	The TRANSIMS Interfaces library.
Source Files	*planio.c*	The plan data structures and interface functions.
	planio.h	The plan interface functions source file.

7.4 Plan File Configuration File Keys

Appendix D provides a description of Plan File configuration file keys.

7.5 Example

Appendix E gives a six-leg plan for traveler 1. The plan consists of a walk-car-walk-bus-walk scenario.

Appendix A: Plan Data Definitions and Format

Column Name	Description	Allowed Values
`Traveler (Person) ID`	Each person is given a unique ID in the population file.	integer
`User Field`	Available to the user to set as desired. Its value is not used internally by the Traffic Microsimulator, but is passed to the output system for use in filtering.	integer
`Trip ID`	Numbers the trips for the traveler sequentially from 1. The trip ID is not used by the Traffic Microsimulator.	unsigned 16-bit integer
`Leg ID`	Numbers the legs within a trip sequentially from 1.	integer
`Activation Time`	The earliest time the simulation needs to worry about this leg. It is generally the starting time (estimated by the Route Planner) for a leg. For a transit leg, however, it represents the arrival time of the passenger at the transit stop, rather than the arrival time of the transit vehicle.	integer: seconds since midnight
`Start Accessory ID`	Denotes the network accessory ID of the starting location for this leg.	unsigned long integer
`Start Accessory Type`	Denotes the type of accessory of the corresponding location. It is necessary because the IDs are not globally unique over accessories. It should be one of: 1) `activity location` 2) `parking` 3) `transit stop` as defined in **`TNetAccessory::EType`** of *NET/Accessory.h*.	integer enumeration
`End Accessory ID`	As above, except it is for the destination rather than the starting accessory.	unsigned long, integer
`End Accessory Type`	As above, except it is for the destination rather than the starting accessory.	unsigned long, integer
`Duration`	In conjunction with `Stop Time` and `Max Time Flag`, specifies how long this leg is expected to take.	integer: seconds
`Stop Time`	In conjunction with `Stop Time` and `Max Time Flag`, specifies an absolute ending time for this leg.	integer: seconds since midnight
`Cost`	Monetary cost of the trip, in cents.	integer

Column Name	Description	Allowed Values
`GCF`	Generalized Cost Function. This is the value that the Route Planner attempts to minimize when planning travelers. Currently the same as duration.	integer
`Max Time Flag`	If true, the end of this activity is best estimated as *max(start time + duration; stop_time)*. Otherwise, use the minimum instead. In the simulation, the actual start time is used, rather than the estimated activation time.	boolean
`Driver Flag`	True, if the traveler is driving a vehicle on this leg.	boolean
`Mode`	Mode of travel. This, together with the driver flag, determines the interpretation of the mode-dependent data following the header. Currently, it must be one of: 0 - `car` 1 - `transit` 2 - `pedestrian` 3 - `bicycle` 4 - `non-transportation activity` 5 - not assigned 6 - `magic move` as defined in the **`TPlan::ETravelMode enum`** of *PLAN/Plan.h*.	integer, enumeration
`Number of Tokens`	Number of white-space-separated tokens in the mode-dependent data block (not including this field itself).	integer

Appendix B: Mode-dependent Data

Table 18. Mode-dependent data for a car driver.

Data	Description	Allowed Values
Vehicle ID	Each vehicle (with its ID) available in the simulation is listed in the vehicle database.	integer
Number of Passengers	The number of passengers, not including the driver, on this leg.	integer
List of Node IDs	The nodes (in order) through which the driver's route will pass.	integer
List of Passenger IDs	The traveler ID of each passenger to be carried on this leg.	integer

Table 19. Mode-dependent data for a car passenger.

Data	Description	Allowed Values
Vehicle ID	Each vehicle (with its ID) available in the simulation is listed in the vehicle database.	integer

Table 20. Mode-dependent data for a transit driver.

Data	Description	Allowed Values
Schedule Pairs	Number of (stop ID, depart time) pairs	Integer
Vehicle ID	Each vehicle (with its ID) available in the simulation is listed in the vehicle database.	Integer
Route ID	Route IDs are specified in the transit route file. Only one route ID is allowed per leg.	Integer
List of Node IDs	The nodes (in order) through which the driver's route will pass.	Integer
List of Schedule Pairs	Each pair consists of a stop ID and a depart time. When a transit driver arrives at a transit stop whose ID is given in this list, the driver will remain at that stop until the depart time.	Integer, integer

Table 21. Mode-dependent data for a transit passenger.

Data	Description	Allowed Values
Route ID	Traveler will board any transit vehicle whose driver's plan matches this Route ID.	integer

Table 22. Mode-dependent data for a pedestrian.

Data	Description	Allowed Values
List of Node IDs	The nodes (in order) through which the traveler's route will pass.	integer

Table 23**. Mode-dependent data for a magic move.**

Data	Description	Allowed Values
Type	Type of magic move plan. 1 – `school bus` 2 – `other`	integer

For activity legs, there is no mode-dependent data.

Appendix C: Route Planner Configuration File Keys

Configuration File Key	Description
ACTIVITY_FILE*	Path to a TRANSIMS activity file.
LOG_ROUTING	Turn on Route Planner logging. This produces information about the status and progress of the Route Planner. Default = 0
LOG_ROUTING_DETAIL	Turn on detailed Route Planner logging. Produces many messages. Default = 0.
MODE_MAP_FILE*	Path to a mode file.
PLAN_FILE*	Name of the file where plans should be written. (Overwrites an existing file.)
ROUTER_BIKING_SPEED	Speed to use when computing delays for walk links traversed by bicycle (meters/second). Default = 4.0
ROUTER_CORR	Floating-point number, between 0 and 1. The Route Planner will change the reported length of a link to be equal to its Euclidean length whenever the ratio of the two is less than this value. This is done in order to avoid problems when the Sedgewick-Vitter heuristic is used. Default = 0.0
ROUTER_DELAY_NOISE	Percentage of noise to add to link delays. Default = 0
ROUTER_DISPLAY_PATHS	If set to 1, list all of the nodes for each leg planned. Note: This produces large amounts of output.
ROUTER_FILTER_EXCLUDE_MODE	Plan modes not include in plan file. Default it to include no modes. Only one of INCLUDE_MODE and EXCLUDE_MODE may be specified.
ROUTER_FILTER_EXCLUDE_VEHICLE	Plan vehicle types not to include in plan file. Default is to include no vehicle types. Only one of INCLUDE_VEHICLE and EXCLUDE_VEHICLE can be specified.
ROUTER_FILTER_INCLUDE_MODE	Plan modes to include in plan file. Default is to include all modes.
ROUTER_FILTER_INCLUDE_VEHICLE	Plan vehicle types to include in plan file. Default is to include all vehicle types.
ROUTER_GET_OFF_TRANSIT_DELAY	Delay encountered when exiting a transit vehicle. Default = 4 seconds
ROUTER_GET_ON_TRANSIT_DELAY	Delay encountered when boarding a transit vehicle. Default = 3 seconds
ROUTER_HOUSEHOLD_FILE	Path to a file containing a list of integer IDs for householders to be planned.
ROUTER_INTERNAL_PLAN_SIZE	Positive integer. Should be enough to accommodate the length (in number of nodes) of the shortest path between any two nodes in the network (and may need to be quite large when multimodal plans are used). Default = 400

Configuration File Key	Description
ROUTER_LINK_DELAY_FILE	Feedback file from which to read link delays. If the configuration file key is not present or the file does not exist, the free speed delays are used.
ROUTER_MAX_NODES_EXAMINED	Maximum number of nodes examined before the Router Planner will conclude that no path exists. Useful mostly for large networks. Default = 400,000
ROUTER_MESSAGE_LEVEL	Level of warning messages to produce: -2 (ERROR) -1 (PRINT) 0 (SEVERE WARNING) 1 (WARNING). Produces information about possible anomalies the Route Planner has encountered. Default = 1
ROUTER_NUMBER_THREADS	Positive integer. Number of worker threads to be used. A value of 0 means no threads will be used. Default = 0
ROUTER_OVERDO	Non-negative floating-point number. If set to 0, no adjustment is made to the distance estimates. If positive, the search for the shortest path to the origin will be biased in the direction of a straight line to the destination. This will produce non-optimal paths. The paths will still be reasonable, but the heuristic may cause relatively small congestion on links to be ignored, and this can break the iterative relaxation mechanism. Default = 0.0
ROUTER_PROBLEM_FILE*	Path name to a file in which activities with anomalies identified by the Route Planner are written.
ROUTER_SEED	Seed to use for random number generator. If the configuration file key is set to 0, use process ID. Default = 0
ROUTER_WALKING_SPEED	Speed to use when computing delays for walk links (meters/second). Default = 1.0
ROUTER_RETIME_PLANS	File containing plans of retimed travelers.
ROUTER_RETIME_TRAVELER_FILE	File containing traveler IDs of travelers to be retimed.
ROUTER_RETIME_MODES	File containing modes to be retimed.
ROUTER_COMPLETED_HOUSEHOLD_FILE	File containing household IDs for plans that have been written to the household file.
TRANSIT_ROUTE_FILE	File containing route of transit vehicles.
TRANSIT_SCHEDULE_FILE	File containing schedules of transit vehicles.
VEHICLE_FILE*	Path to a TRANSIMS vehicle file.

*Required.

Appendix D: Plan File Configuration File Keys

Configuration File Key	Description
CA_USE_PARTITIONED_ROUTE_FILES	If this configuration file key is set, the Traffic Microsimulator expects to find separate indexes into a plan file for each slave. These can be produced using a partition file and the *DistributePlans* utility.
PLAN_FILE	Location of a file containing plans, or the base name of an index that points to plan files. Used by the Route Planner for output and the Traffic Microsimulator and Selector/Iteration Database for input.

Appendix E: Annotated Example of a Plan

Trip/Leg	Plan	Description

Trip 1/Leg 1	`1 156 1 1 1 0`
	`25200 123 1 456`
	`33 25200 1`
	`0 2`
	`2`
	`1000 1001`
		The user has chosen to mark this leg with the code 156, which has meaning only to that user but will be duly reported in any output concerned with this leg. It is trip 1, leg 1 for this traveler. It is the first leg to be simulated for this traveler, but not the last. The Route Planner expects the trip to start at 25200 = 7∗3600 = 7 AM. The leg will start at activity location 123 and end at parking accessory 456. The Route Planner expects the trip to take 33 seconds. The traveler's next leg will begin upon arrival at the destination or 33 seconds after departure from the origin, whichever is later. The traveler is not driving a vehicle and is, in fact, walking (mode = 2). There are two tokens of mode-dependent data, which in this case might be the nodes traversed. The Traffic Microsimulator would probably simply use the planner's estimated duration and place the traveler in the destination queue 33 seconds after his arrival at the origin. However, the Traffic Microsimulator could also choose to estimate its own duration. The Traffic Microsimulator will not use the node information.

Trip 1/Leg 2	`1 156 1 2 0 0`
	`25233 456 2 789`
	`1314 0 1`
	`1 0`
	`18`
	`0 0`
	`1 2 3 4 5 6 7 8 9`
	`10 11 12 13 14 15 16`
		Leg 2 of trip 1 is neither the first nor the last. The traveler will be driving (driver flag = 1) a car (mode = 0) from parking accessory 456 to parking accessory 789 via the 16 nodes 1-16 using vehicle 0, carrying no passengers. The expected start time is 7:00:33 a.m., and the expected duration is 1314 seconds.

Trip 1/Leg 3	`1 156 1 3 0 0`
	`26547 789 2 10 2`
	`127 0 1`
	`1 0`
	`5`
	`0 1`
	`17 18`
	`1000`
		Traveler 1 picks up one passenger (traveler 1000) and drives to parking accessory 10 via nodes 17 and 18.

Trip 1/Leg 4	`1 156 1 4 0 0`
	`26674 10 2 11 3`
	`30 0 1`
	`0 2`
	`0`
		The traveler walks (mode = 2) from parking accessory 10 to bus stop (accessory type = 3) 11. The Route Planner, knowing that the Traffic Microsimulator will not simulate walking, has chosen not to write out the details of the path the walker will take (Number Of Tokens = 0).

Trip 1/Leg 5	`1 156 1 5 0 0`
	`26704 11 3 4 3`
	`1502 0 1`
	`0 1`
	`1`
	`72`
		The traveler will ride in (driver_flag = 0) the first bus (mode = 1) arriving on route 72, from bus stop 11 to bus stop 4.

Trip 1/Leg 6	`0 156 1 0 0`
	`28206 4 3 5 1`
	`31 0 1`
	`1 2`
	`0`
		The traveler takes 31 seconds to walk from bus stop 4 to activity location 5.

Trip 2/Leg 1	`1 156 2 0 1`
	`28237 5 1 5 1`
	`28800 61200 1`
	`1 4`
	`0`
		This is the first leg of trip 2 for traveler 1. Since the last leg flag is set, it is also the last leg that will be simulated. It is an activity (mode = 4) that ends at 5:00 p.m. (= 17 ∗ 3600 = 61200 seconds) or eight hours (= 8 ∗ 3600 = 28800) after arrival, whichever is later. There is no data associated with this leg, although the Route Planner could, in principle, add anything—a list of projects the person will be working on, a list of groceries to buy, etc.

Appendix F: Error Codes

Error codes for the Route Planner are in the range 25000 – 25999.

Table 24. Route Planner error codes.

Code	Description
25001	Couldn’t read activity file.
25002	Couldn’t read household file.
25003	Couldn’t read mode map file.
25004	Invalid program arguments.
25005	Required configuration file key not specified.
25006	Standard exception caught.
25007	Unknown exception caught.

Chapter Four: Index